2INC0 - Operating Systems

I/O management

Geoffrey Nelissen

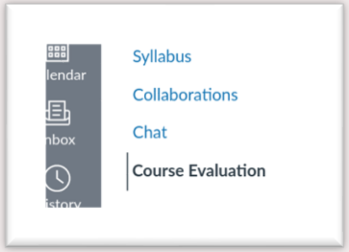
IRiS
Interconnected
Resource-aware
Intelligent Systems

TU/e Technische Universiteit Eindhoven University of Technology
Where innovation starts

Announcement:
The course evaluation surveys are open

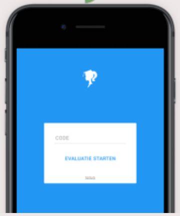
Where can you give feedback?

Canvas tab



Personal email invite

evalytics



https://app.evalytics.nl

Feedback on the final exam/assignment

The course evaluation surveys will be closed one day before the final exam; if you want to give feedback about the final exam/assignment, you can use this link or QR code.

Questions about anything regarding quality assurance?
Mail us at ***mcs.quality.assurance@tue.nl***

You can use these tips to provide more impactful feedback

Be **specific** and **focused** on your feedback. Use examples and **suggestions** to avoid vague statements.

Always be **respectful** when giving feedback.

Give positive feedback as well as areas for improvement. Stay **solution-oriented**.

When giving feedback, focus on the **1 or 2** most important points that apply to you.

What happens with the results of the surveys?

The responsible teacher reads the anonymous results and reflects on their course.

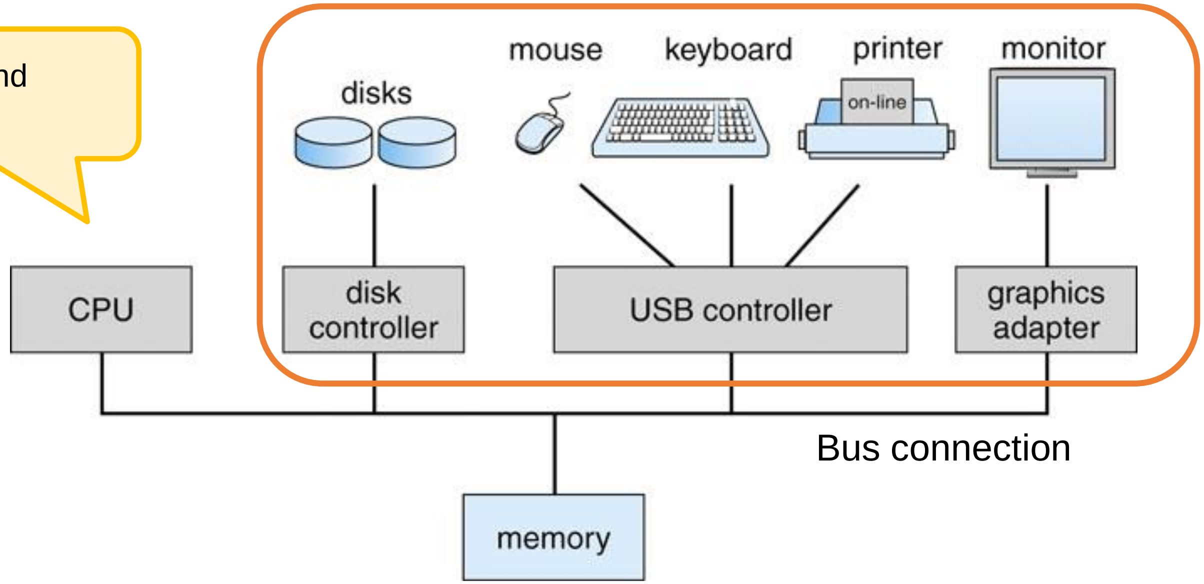
Results and reflection are discussed during committee meetings to **improve the courses and programs!**

Computer system

Allows to interact with the physical world / users / other machines

Execute OS and programs

Inputs/outputs



Bus connection

Processes (i.e., program in execution) need to be in main memory to be executed

Course Overview

- **Introduction to operating systems** (lectur

Purpose of an operating system

- **Processes, threads and scheduling** (lectures 2+3)

Structure of running programs.
How to **implement concurrency.**
How to decide **what to execute and when.**

- **Concurrency and synchronization**
  - atomicity and interference (lecture 4)
  - actions synchronization (lecture 5)
  - condition synchronization (lecture 6)
  - deadlock (lecture 7)

Problems associated to **concurrent executions.**
How to prove program properties **(topology invariants, traces)**
How to protect **critical sections.**
How to **synchronize the execution of** programs to enforce new properties
How to analyze **deadlocks**, prevent them and detect them.

- **File systems** (lecture 8)

How files are **organized (virtually)**
How can they **be accessed (physically on hard drive)**

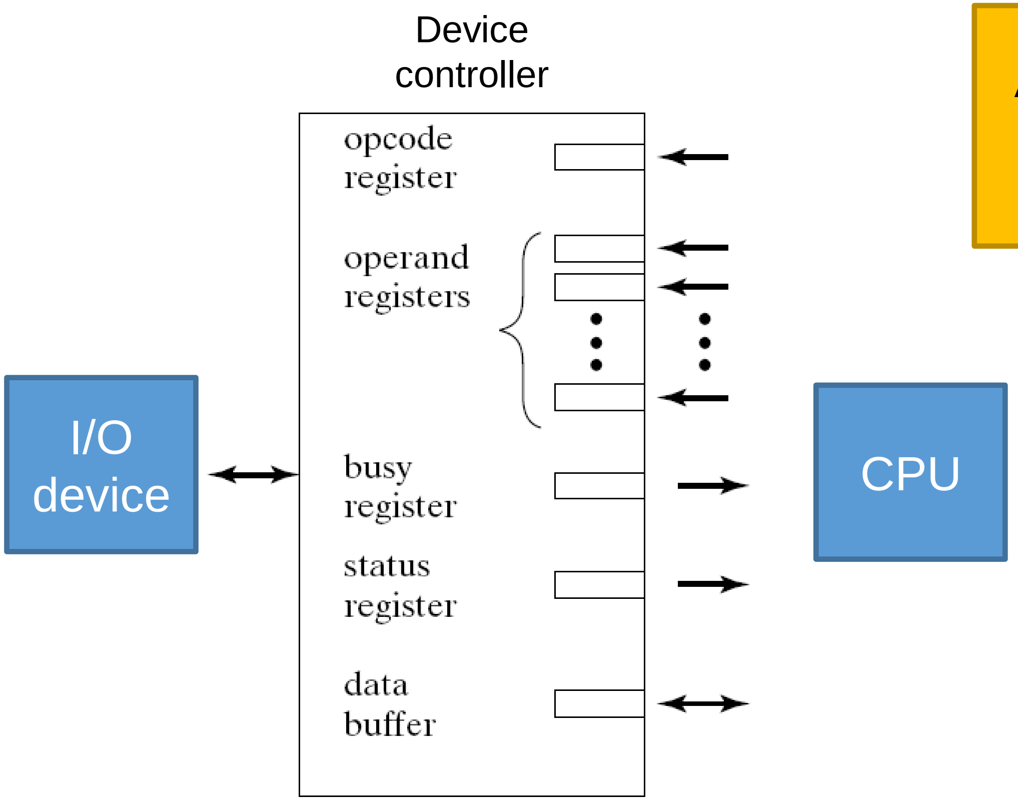
- **Memory management** (lectures
- **Input/output** (lecture 12)

How **to efficiently load** processes in main memory.
How to efficiently manage **limited physical memory space**.
How to **share memory space** between concurrent processes.

Agenda

- I/O device controllers
- I/O subsystem
- I/O buffering
- Disk scheduling

I/O controller interface example



An I/O device **communicates** with the CPU **through hardware registers** available **in a device controller**

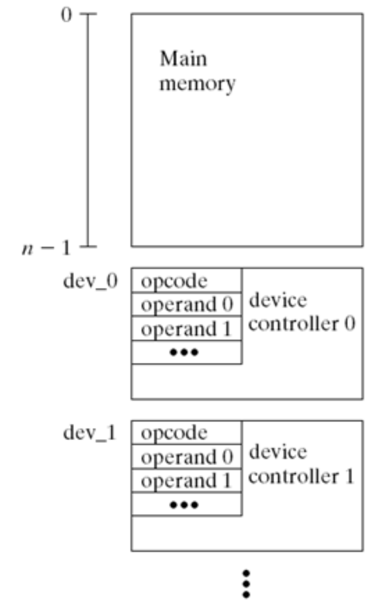
- Opcode register: code of the operation to be performed, e.g. read, write
- Operand registers: parameters associated with the operation
  - e.g. addresses to be red from or written at, DMA parameters, etc.
- Busy and status registers: provide information about availability, readiness, errors
- Data buffer registers: bytes transferred to or from the I/O device
  - e.g. value typed in on the keyboard, or text to be printed

Two methods for accessing the device controller

Method 1

The CPU instruction set is extended with special I/O instructions

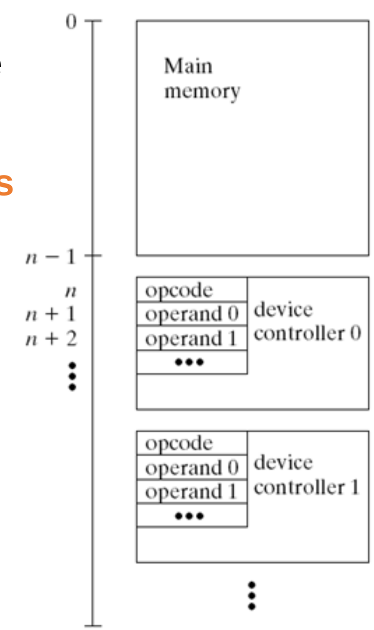
- **Example** of assembly instructions:
  - **io_store** cpu_reg, dev_no, dev_reg
  - **io_load** dev_no, dev_reg, cpu_reg
- No physical address associated to each register
  - ➔ **Advantage**: no interference with virtual memory
  - ➔ **Disadvantage**: not possible to map device in user space
    - ➔ user process cannot directly access the device as a normal data structure.



Method 2

Physical address space is extended to directly refer to device registers

- Each device controller register is given a physical address
  - ➔ **Advantage**: the device may be mapped to the user space (i.e., we associate a virtual address to the physical address of each I/O register).
  - ➔ **Disadvantage**: complexify virtual memory management



Two methods for accessing the device controller

Method 1

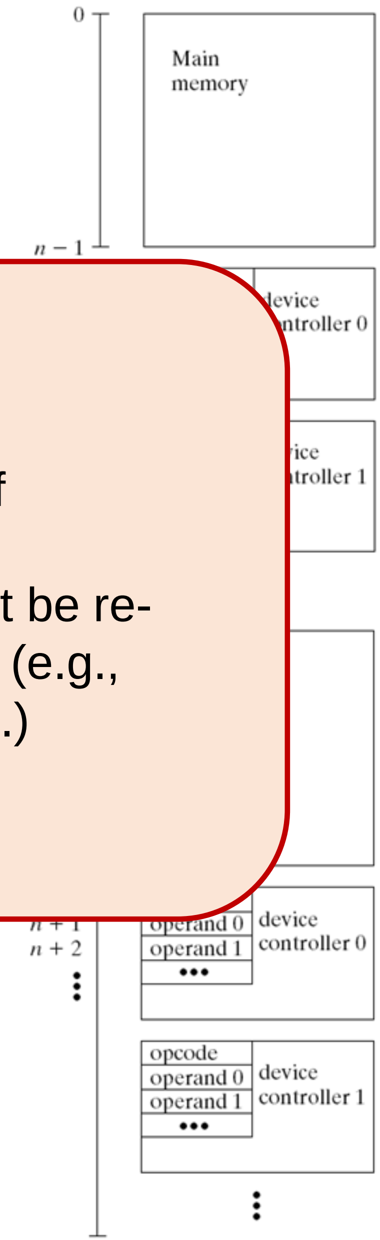
The **CPU instruction set is extended** with special I/O instructions

- **Example** of assembly instructions:
  io_store cpu_reg, dev_no, dev_reg
  io_load dev_no, dev_reg, cpu_reg

Issue:

- **Each I/O device controller** may have a **different** set of registers/opcodes/operands
  ➔ Code is written for a specific I/O controller and must be re-written if we change the type or brand of I/O device (e.g., move from HDD to SSD, change brand of SSD, etc.)
  ➔ **Limits portability**
  ➔ **Increases work, risk of bugs, etc.**

➔ **Disadvantage**: **complexify virtual memory management**



Agenda

- **I/O device controllers**
- **I/O subsystem**
- **I/O buffering**
- **Disk scheduling**

I/O subsystem

- Part of the OS that manages input and output devices

- **Goals**:
  - to present a **logical/abstract view** of I/O devices
  - to **facilitate sharing** of devices
  - to provide **efficiency** and **optimize performance**
    - Examples:
      - ensures the CPU and multiple I/O devices run in parallel
      - re-ordering I/O requests to improve throughput
      - buffering to hide latency
      - …

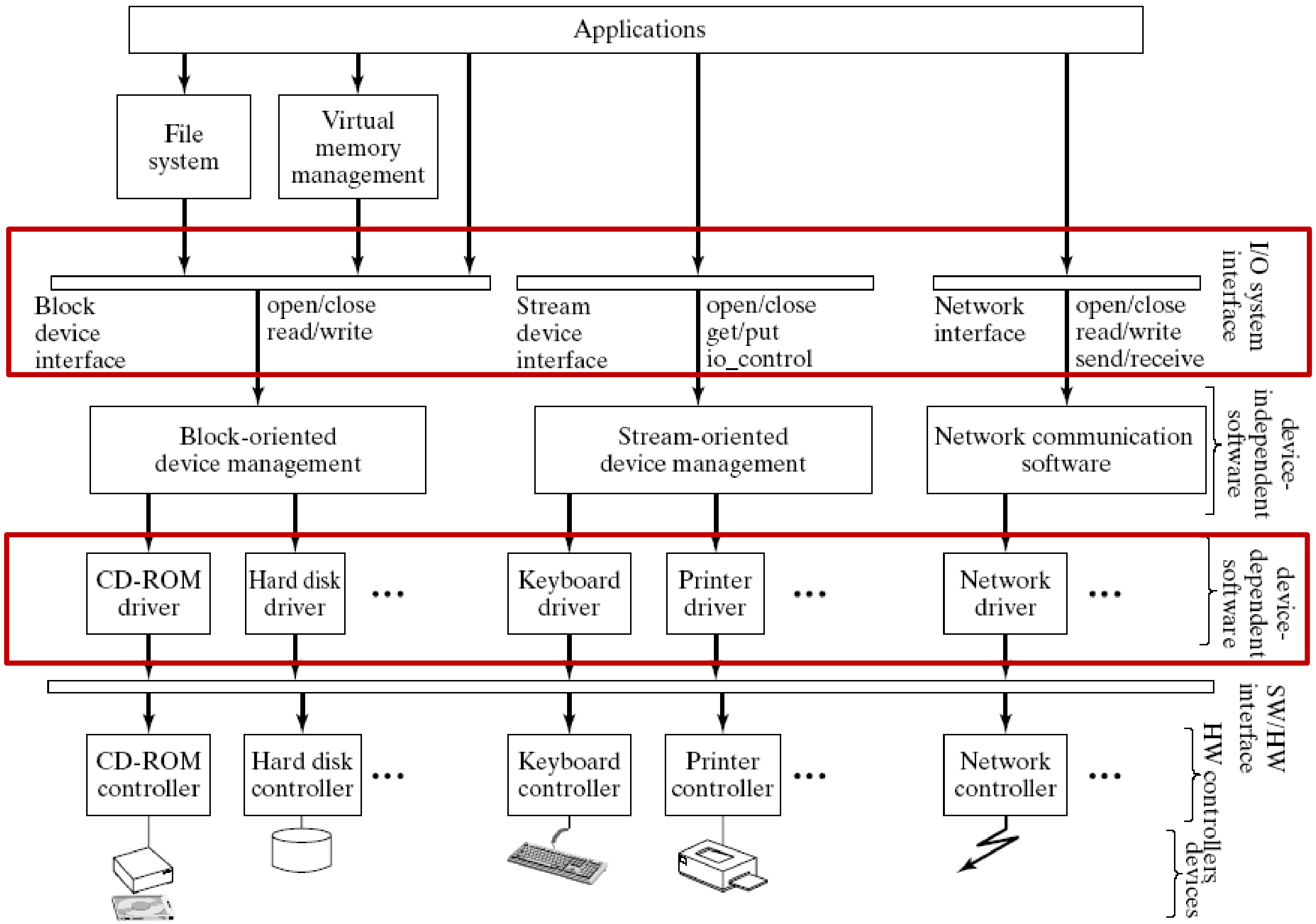
Design concerns:

- large **variety** of I/O device types
  - keyboard, display, printer, disk, temperature sensor, network cards ....
- **different speeds** and **(brand-)specific approaches**
- ensure we can **add new devices** after the OS development and installation

Two levels of abstraction

High level I/O abstraction (for the user)

Low level I/O abstraction (for the OS)



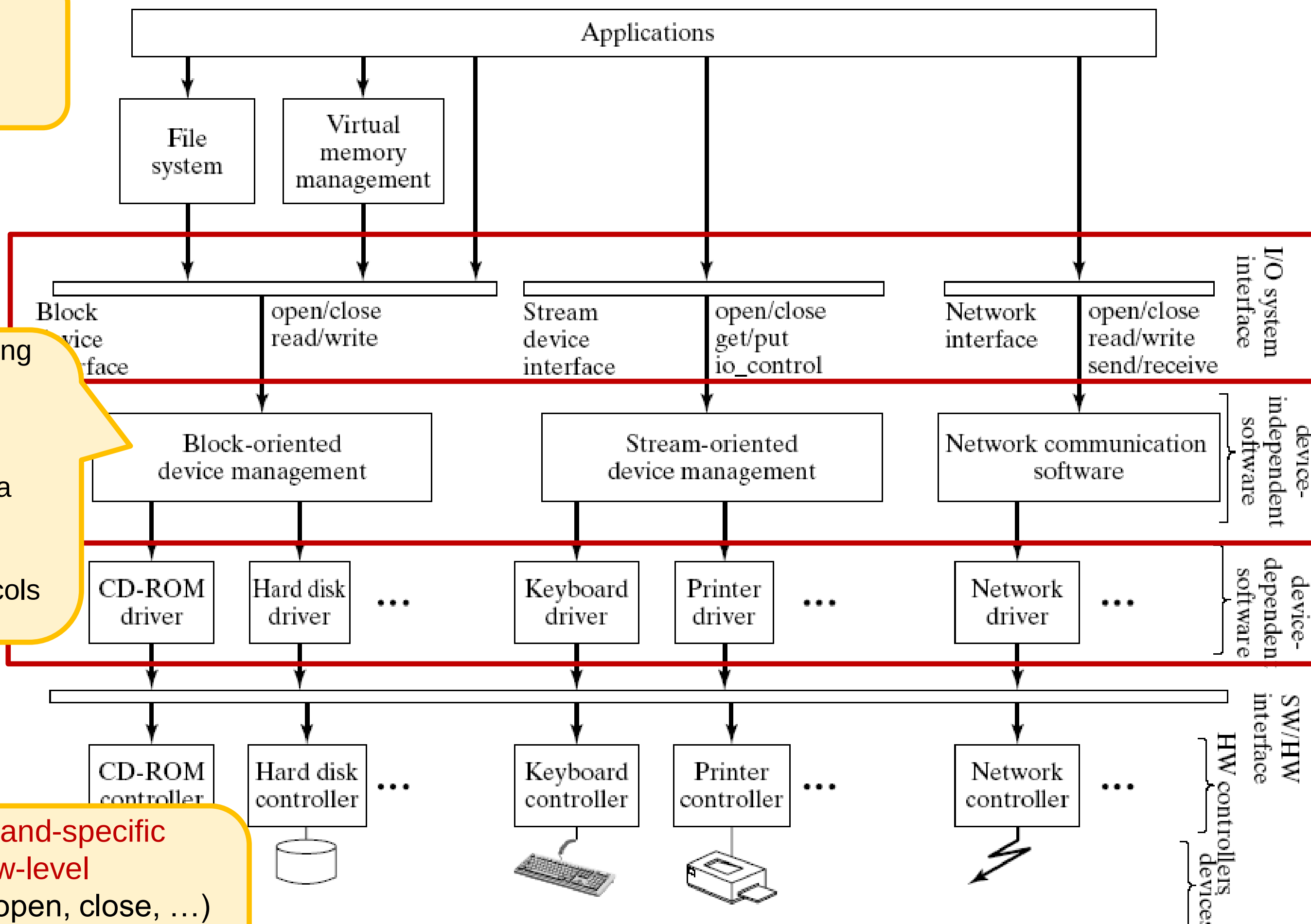
Reference: Bic, Lubomir, and Alan C. Shaw. *Operating systems principles*. Prentice Hall, 2003.

Two levels of abstraction

User level abstraction: standardized APIs defined per device class

High level I/O abstraction

Provide generic code for handling **classes**/families **of I/O devices**
Examples:
- Sequence of high-level operations to read or write data on disk/cd-rom/…
- implementation of specific network communication protocols (e.g., TCP)



Low level I/O abstraction

OS level: encapsulation of brand-specific issues behind well-defined low-level operations (e.g., read, write, open, close, …)

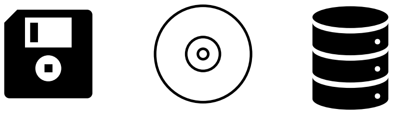
For each specific device, a **device driver implements those low-level operations**

Reference: Bic, Lubomir, and Alan C. Shaw. *Operating systems principles*. Prentice Hall, 2003.

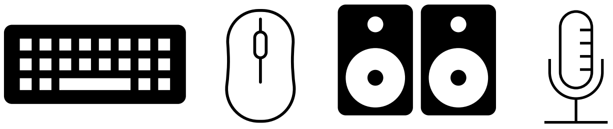
Agenda

- I/O device controllers er
- I/O subsystem
  - High level abstraction
  - Low-level abstraction
- I/O buffering
- Disk scheduling

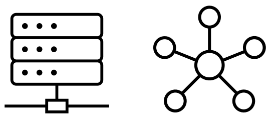
Divide I/O devices in different classes



- **Block-oriented** or storage devices
  - Read and write blocks of **data in arbitrary order**
    i.e. readers/writers model
    ➔ **Each block of data has an address that can be accessed at any time**
  - **Sequence of actions that must be performed to access data depends on** the **operation** (e.g., read/write/copy), **data location**(s), and **last** (set of) **data accessed before**
  - examples: hard drive, ssd, CD reader, magnetic tape



- **Stream-oriented** devices
  - input and output **data in a sequential way**
    i.e. producer/consumer model
  - **Optimized for single byte access**
  - examples: keyboard, sensors, actuators, mouse



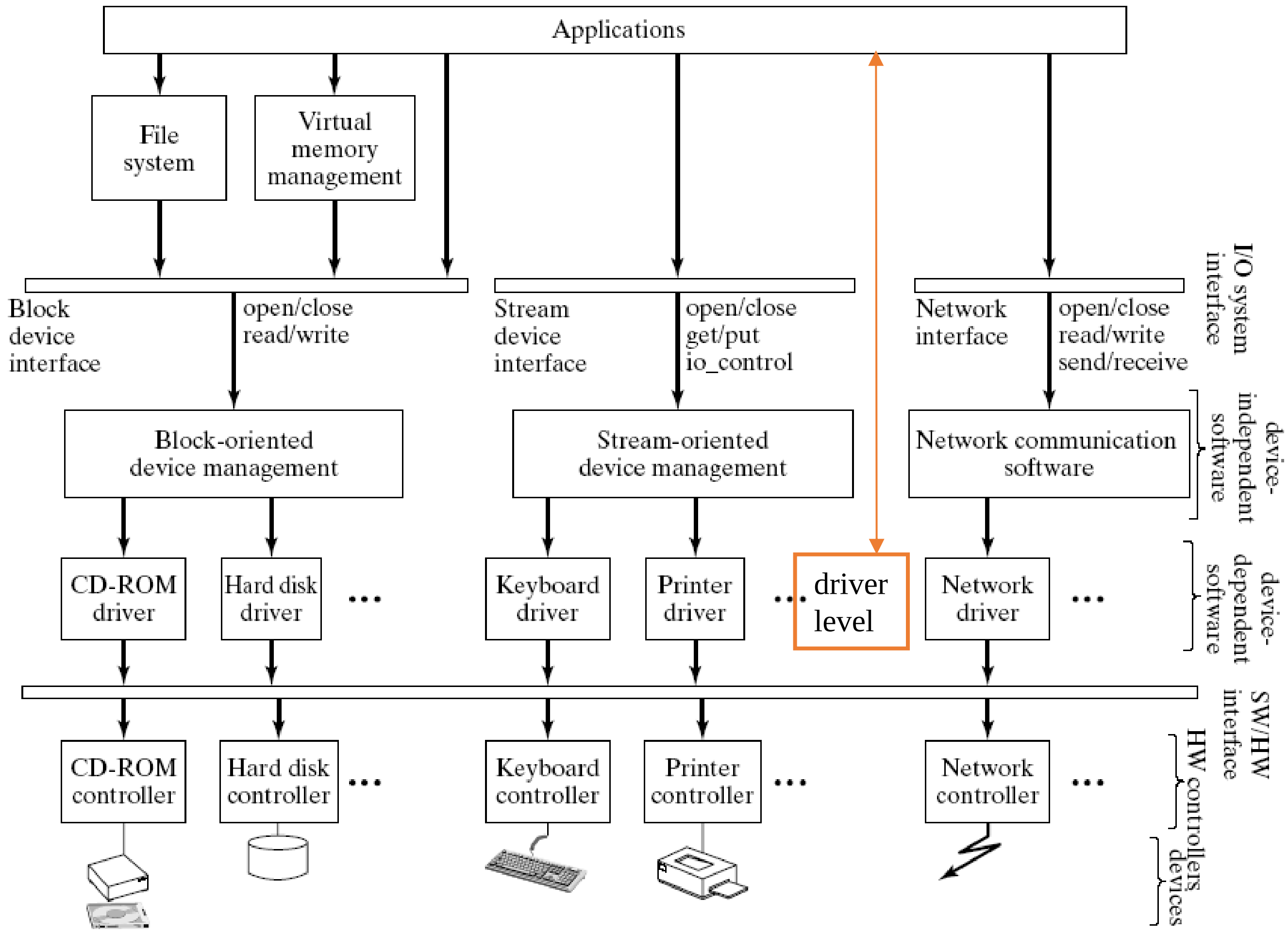
- **Network communication** devices
  - send and receive packets on a network (socket) interface
  - Provide connection and communication protocols

Note: the above **'classical classification' may not be enough**

- **Example**: accessing graphics hardware not really covered by the standard interface: read() / write()
  ➔ very low performance with classical I/O system interfaces

Two main solutions when I/O devices do not fit in any class

- **Solution 1**: let the user application directly access the driver



Two main solutions when I/O devices do not fit in any class

- **Solution 1**: let the user application directly access the driver
  - **Drawbacks**:
    - Application code becomes device specific
    - Application code operates at a low level of abstraction

- **Solution 2**: extend the capabilities of the OS with **new APIs** via external libraries to
    - provide an ***abstraction*** from vendor-specific issues
    - support ***domain-specific concepts*** needed for device-independent application development
      Examples: audio sink, filter operations, graphics scene, ...
    - provide ***optimized performance***

    - **Examples: Microsoft DirectX or OpenGL for graphic or CUDA for GPGPU management**

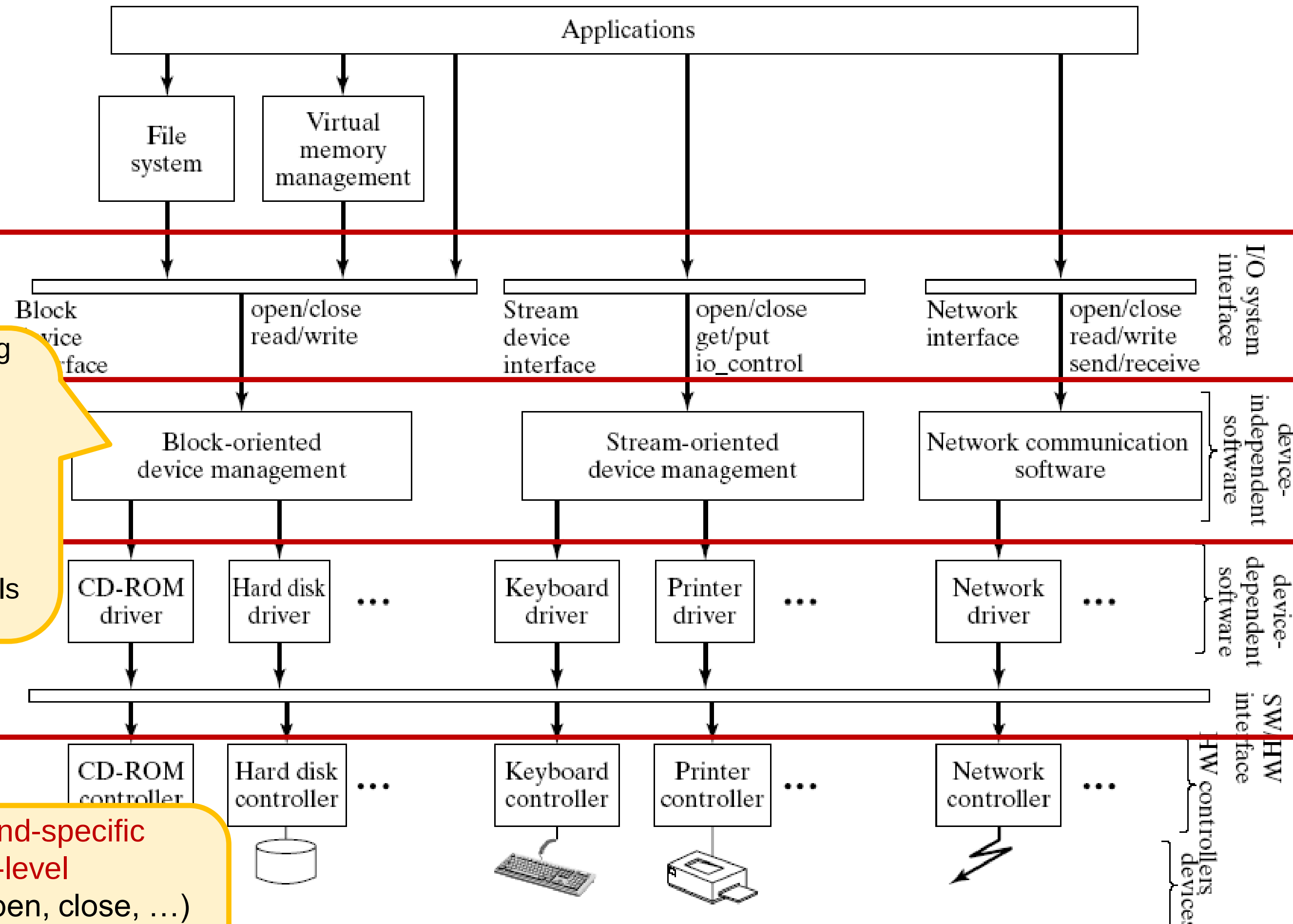
Agenda

- I/O device controllers
- I/O subsystem
  - High level abstraction
  - Low-level abstraction
- I/O buffering
- Disk scheduling

Two levels of abstraction

TU/e

High level I/O abstraction



Provide generic code **for** handling **classes**/families **of I/O devices**
Examples:

- Sequence of high-level operations to read or write data on disk/cd-rom/…
- implementation of specific network communication protocols (e.g., TCP)

Low level I/O abstraction

OS level: encapsulation of brand-specific issues behind well-defined low-level operations (e.g., read, write, open, close, …)

For each specific device, a device driver **implements those low-level operations**

ence: Bic, Lubomir, and Alan C. Shaw. *Operating systems principles*. Prentice Hall, 2003.

IRIS

Device driver interface

- The device driver implements a **collection of standard operations** (functions)
  - in Linux: the functions (say, for device xxx) are registered in a data structure

```
struct file_operations xxx_fops = {
    NULL,          /* lseek()   */
    xxx_read,      /* read()    */
    xxx_write,     /* write()   */
    NULL,          /* readdir() */
    NULL,          /* select()  */
    xxx_ioctl,     /* ioctl()   */
    NULL,          /* mmap()    */
    xxx_open,      /* open()    */
    xxx_close      /* close()   */
    };
```

- *read*: function to read data
- *write*: function to write data
- *lseek*: move the read/write pointer
- *ioctl*: i/o control to modify device/driver parameters
- *select*: notify when the i/o device is ready to perform a specific operation
- …

Driver communication with the device controller

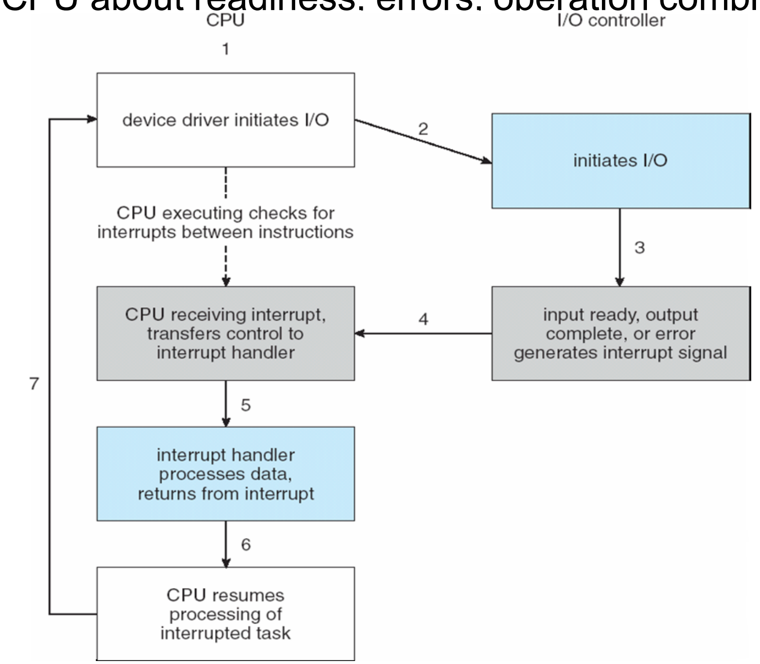
- **Polling**
  - Driver initiates I/O operations in the device controller and observes completion (i.e., driver busy-waits) or periodically wakes-up to check completion
  - The driver polls the device controller repeatedly and tests
  - **Wastes CPU time**

When is it acceptable to busy-wait?

- Only **when I/O operations are** known to be **fast in comparison to** the overhead of **context switches**

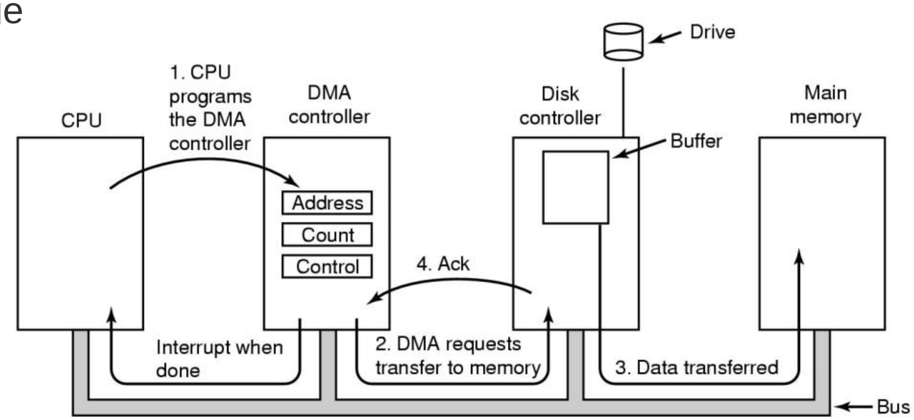
Driver communication with the device controller

- **Polling**
  - Driver initiates I/O operations in the device controller and observes completion (i.e., driver busy-waits) or periodically wakes-up to check completion
  - The driver polls the device controller repeatedly and tests
  - **Wastes CPU time**
- **Interrupts**
  - Driver initiates I/O operations in the device controller and then **yields the CPU**
  - Device controller uses interrupts to inform the CPU about readiness, errors, operation completion, …



Driver communication with the device controller

- **Polling**
  - Driver initiates I/O operations in the device controller and observes completion (i.e., driver busy-waits)
  - The driver polls the device controller repeatedly and tests
  - **Wastes CPU time**
- **Interrupts**
  - Driver initiates I/O operations in the device controller and then **yields the CPU**
  - Device controller uses interrupts to inform the CPU about readiness, errors, operation completion, …
- **DMA**
  - DMA can be implemented inside the device controller or as a separate hardware component
  - After initialization, the DMA independently **moves groups of data** between the device controller and memory
  - **less overheads for the CPU** only one interrupt handling when the whole transfer is completed instead of one interrupt per word/message



Reference:Tanenbaum, Andrew. Modern operating systems. Pearson Education, Inc.,, 2009.

Agenda

- I/O device controllers
- I/O subsystem
  - High level abstraction
  - Low-level abstraction
- I/O buffering
- Disk scheduling

I/O buffering, motivation

- As an example, consider a process reading or writing in a file located on a hard drive
  - It issues the command
  - then the process is either waiting or suspended until an interrupt

- **Potential problems**
  - **Speed / latency mismatch**:
    Process must wait for the relatively slow I/O to complete before it can send new data to write on the disk
  - **Data granularity mismatch** (byte vs line vs block):
    Application may expect to receive data in smaller pieces than a block (and vice versa)
  - **Conflict with the swapping decisions made by the OS**:
    Pages containing the virtual address range must remain in physical memory until I/O completes (otherwise, the driver/DMA may write the data at the wrong physical address or corrupt the address space of another process)

I/O buffering, motivation

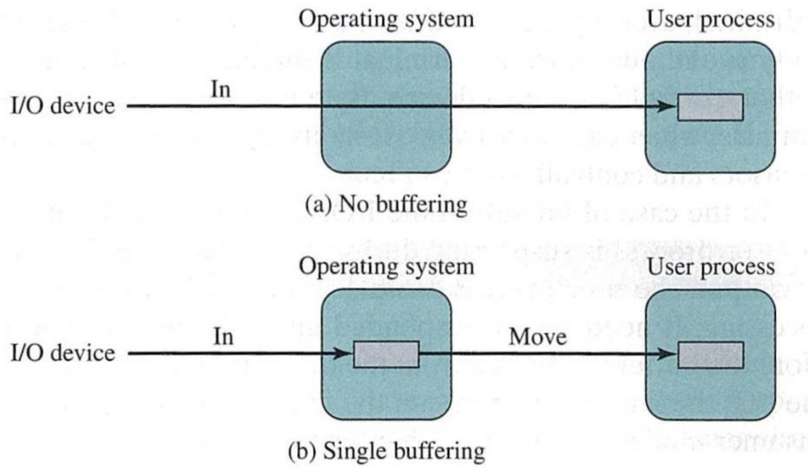
Buffer

dedicated (kernel) memory space or hardware registers that holds data of a producer until its consumer is ready to consume

Main task: decouple producer and consumer

- resolve **speed difference** and latency problems
- resolve **granularity differences**
  - driver returns characters, application needs lines
  - driver returns blocks, application needs bytes
- resolve **swapping issues**
  - The buffer remains permanently in physical memory even when the process is swapped out
- **improve efficiency**
  - Perform input transfers before request: try to **predict what will be needed in the future**
  - Delay outputs on purpose: wait for the right time to perform output (e.g., more data can be transferred at once, or optimize seek time (**see disk scheduling**))
  - Caching data

Buffering alternatives



picture from Stallings,
Operating Systems – Internals and Design Principles

Buffering inputs:

The data is written into the buffer first (e.g., from disk to kernel memory address space).

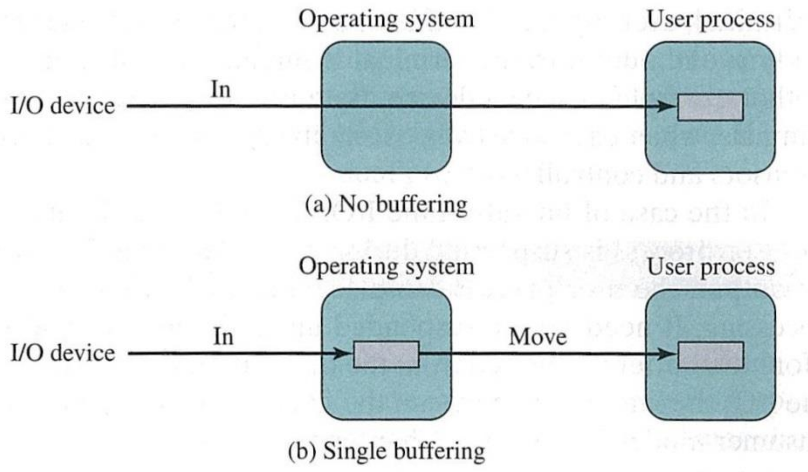
Then, the data is moved into the user process address space.

Buffering outputs:

The data is moved to the buffer first (e.g., from user space to kernel space).

Then, data is output (e.g., written to a disk) directly from the buffer.

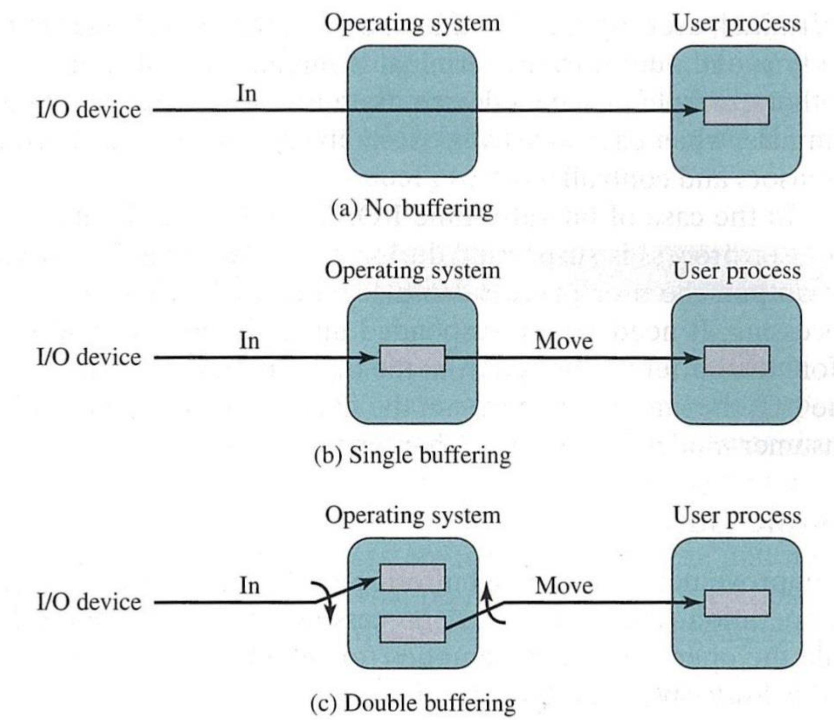
Buffer use schemes for producer/consumer



picture from Stallings,
Operating Systems – Internals and Design Principles

- **Single buffering**
  - **enables asynchronous transfer**
    - The input device controller can produce data without waiting for the process to be ready to consume it.
      ➔ releases I/O device to perform another operation
    - The process can buffer output data even if the device controller is not yet ready to transfer it.
      ➔ allows process to continue doing something else
  - **allows swapping the process out of main memory**

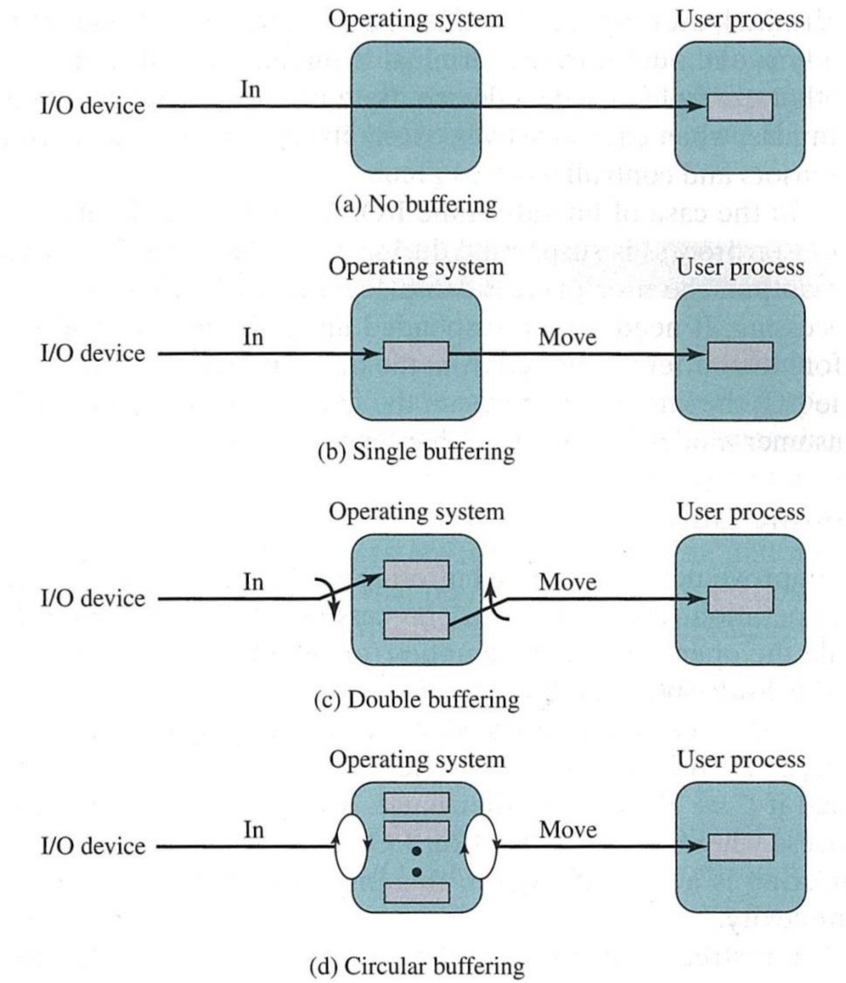
Buffer use schemes for producer/consumer



- **Double buffering** (buffer swapping)
  - reduces idle time
    - OS can move the content of one buffer from kernel to user space while the device controller is filling the other buffer
  - still poor in handling bursts of data

picture from Stallings,
Operating Systems – Internals and Design Principles

Buffer use schemes for producer/consumer



picture from Stallings,
Operating Systems – Internals and Design Principles

- **Double buffering** (buffer swapping)
  - reduces idle time
    - OS can move the content of one buffer from kernel to user space while the device controller is filling the other buffer
  - still poor in handling bursts of data
- **Circular buffering**
  - to handle bursts of data

Evaluating buffering performance

Throughput

- How many data can be transferred per second?
  (we limit ourselves to analyzing the input scenario)

Throughput calculation

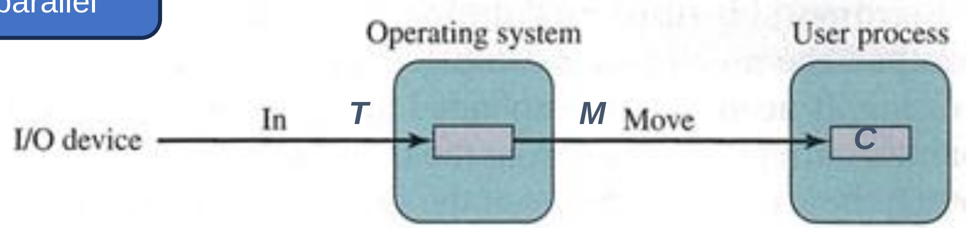
T: time to **transfer** one data in main memory
M: time for **moving** one data from kernel address space to user address space
C: time the process need to operate on one data (**computation time** on received data)
D: Minimum time until the next data may become available in user space

- **Throughput** = 1/*D*
  - no buffer: $D = T + C$

    (because M=0 in this case, and we cannot transfer a new data until the previous data is processed)
  - single buffer: $D = max\ (C,T) + M$

    (because we can move a data between kernel and user space only when a new data is transferred in the buffer and the previous data has been processed, but the computation may happen in parallel to the data transfer in the buffer)

Because T and C can happen in parallel

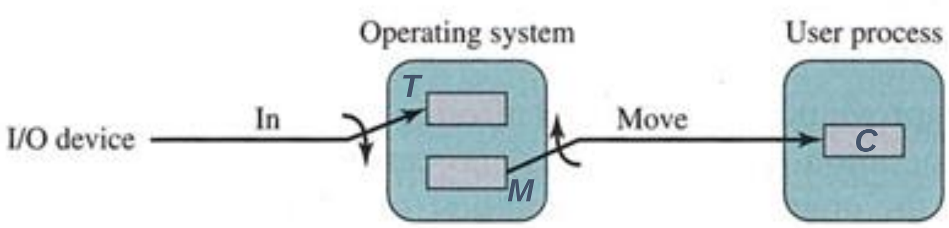


(b) Single buffering

Throughput calculation

T: time to **transfer** one data in main memory
M: time for **moving** one data from kernel address space to user address space
C: time the process need to operate on one data (**computation time** on received data)
D: total time until a data is available in user space

- **Throughput** = 1/*D*
  - no buffer: $D = T + C$ *(because M=0 in this case, and we cannot transfer a new data until the previous data is processed)*
  - single buffer: $D = max\ (C,T) + M$ *(because we can move a data between kernel and user space only when a new data is transferred in the buffer and the previous data has been processed, but the computation may happen in parallel to the data transfer in the buffer)*
  - double buffer: $D = max\ (M+C,\ T)$ *(because the move cannot start before the computation completes, but the move and computation happen in parallel to the data transfer)*



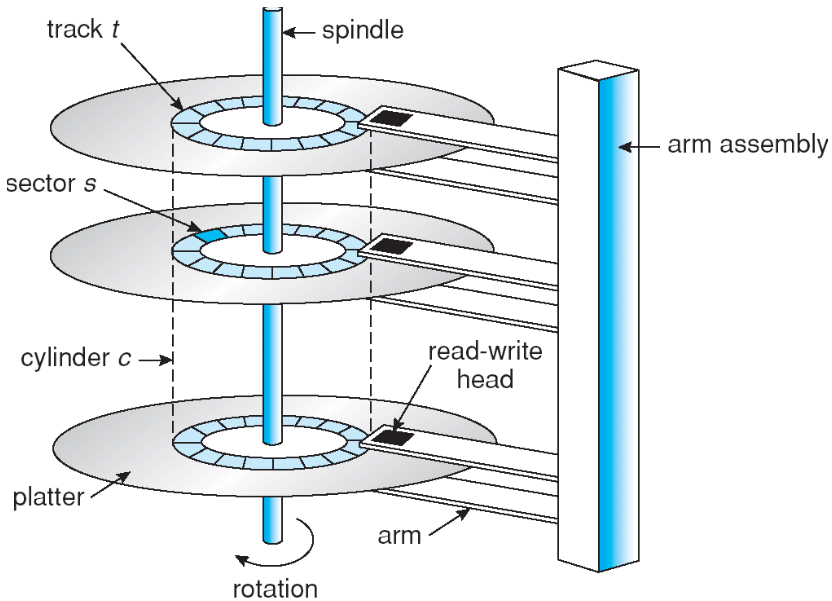
(c) Double buffering

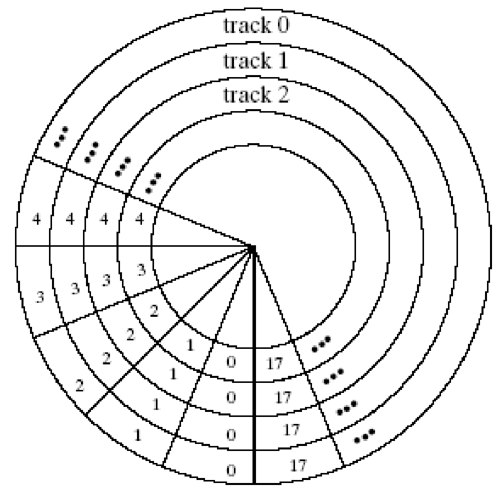
Agenda

- I/O device controllers
- I/O subsystem
  - High level abstraction
  - Low-level abstraction
- I/O buffering
- Disk scheduling

Hard-drive operation

- Several **platters**, divided in **tracks**, divided in **sectors**
- **Cylinder**: set of tracks that are at the same arm position (same radius).





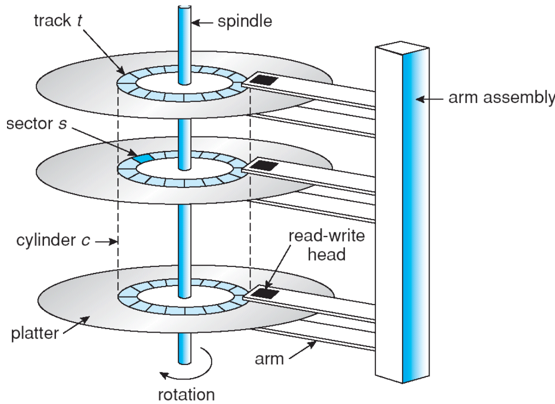
Hard-drive operation

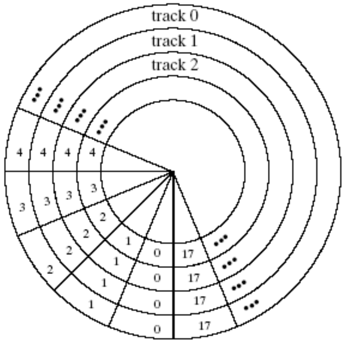
- **Average time to read/write one block on the disk**

$$T_{read} = T_{seek\ cylinder} + T_{move\ to\ sector} + T_{read\ block}$$

$$= T_{seek\ cylinder} + \frac{1}{2r} + \frac{B}{rN}$$

r = rotations per second
N = #bytes per track
B = #bytes per block/sector





- Reading/writing a block of data in a sector:
  - **head-positioning time on correct cylinder (= seek time)**
    - On average: 9-12 msec
    - ~1-2 msec for neighbouring tracks
  - **read/write within a track** (depending on rpm): ~3-5ms

➔ The OS must optimize the disk head movement

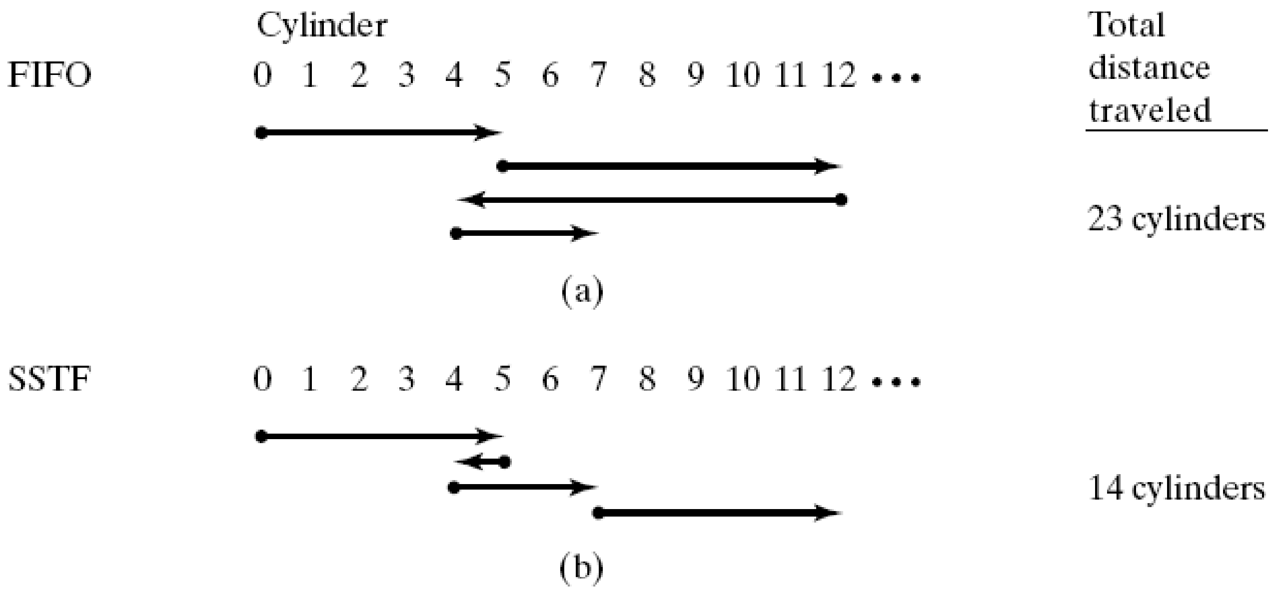
Disk scheduling

- Requests for blocks are buffered.
- The block requests in the buffer are treated according to a scheduling policy:
  - **FIFO**
    - treats requests in the same order they are submitted
  - **Shortest Seek Time First**
    - treats the buffered request that requires the smallest head movement

    ➔ possible starvation, unpredictable
  - **Elevator Scan**
    - completes full swing of the head in either direction
  - Many other versions investigated in the literature

Comparison on an example

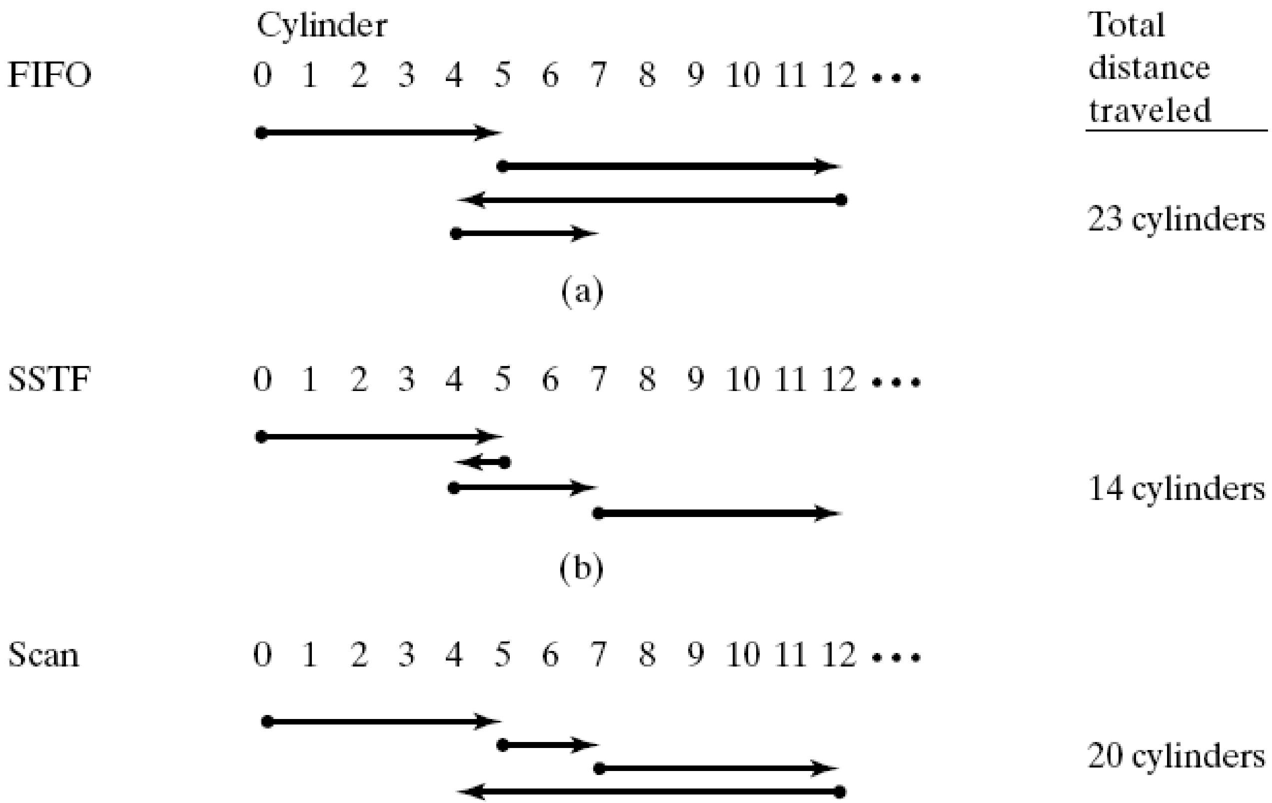
Assume we just moved from cylinder 0 to cylinder 5, and that the requests in the buffer are requesting accesses to cylinders 12, 4 and 7.



Notes:
- SSTF may lead to **starvation**

Comparison on an example

Assume we just moved from cylinder 0 to cylinder 5, and that the requests in the buffer are requesting accesses to cylinders 12, 4 and 7.



Summary

- **The I/O subsystem adopts a layered approach** based on **two levels of abstraction**
  - User interface
  - Driver interface

- **I/O buffering** decouples producers from consumers
  - Addresses speed, granularity, process swapping and efficiency issues

- **Disk scheduling**: example of how I/O buffering can be used to **reorder I/O transfer requests and improve efficiency**

- **Deadline of Homework 5 at the end of the week**
- **One more homework on file systems will be released**
- **Q&A session next Friday at 9:45**
- **Send me your questions before the session for more complete answers**

- **Course survey is open**